(2)

AVF Control Number: NBS87VDDC525_1

Ada Compiler
VALIDATION SUMMARY REPORT:
Certificate Number: 871125S1.09003
DDC-I, Inc.
DACS-80x86, Version 4.2
Host: DEC MicroVAX II
Targets:
Intel 8086 iSBC 86/05A 1MB memory (bare microprocessor)
Intel 80186 iSBC 186/03A 1MB memory (bare microprocessor)
Intel 80386 iSBC 386/21 1MB memory (bare microprocessor)
Titan 8086 SECS 86/20 640KMB memory (bare microprocessor)
Titan 80286 SECS 286/20 640KMB memory (bare microprocessor)

Completion of On-Site Testing:
25 Nov 1987

Prepared By:
Software Standards Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ADA190013

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETEING FORM

1. REPORT NUMBER | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Ada Compiler Validation Summary Report: DDC-I, Inc., DACS-80x86, Version 4.2, DEC MicroVAX II (Host) and Intel 80x86 bare processors (Targets).

5. TYPE OF REPORT & PERIOD COVERED
25 Nov 1987 to 25 Nov 1988

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

National Bureau of Standards,
Gaithersburg, Maryland, U.S.A.

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

National Bureau of Standards,
Gaithersburg, Maryland, U.S.A.

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE
25 November 1987

13. NUMBER OF PAGES
73 p.

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)

National Bureau of Standards,
Gaithersburg, Maryland, U.S.A.

15. SECURITY CLASS (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DACS-80x86, Version 4.2, DDC-I, Inc., National Bureau of Standards, DEC MicroVAX II (Host) under MicroVMS, Version 4.4, and Intel 8086, 80186, 80386, Titan 8086, 80286 (Targets) bare microprocessors, ACVC 1.9.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

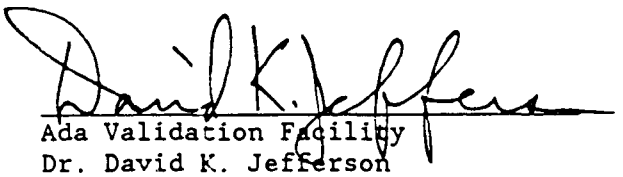
Compiler Name: DACS-80x86, Version 4.2

Certificate Number: 871125S1.09003

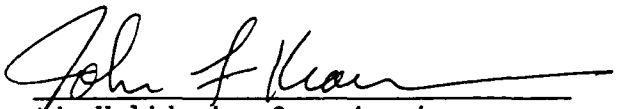
Host:
DEC MicroVAX II under MicroVMS, Version 4.4
Targets:
Intel 8086 iSBC 86/05A 1MB memory (bare microprocessor)
Intel 80186 iSBC 186/03A 1MB memory (bare microprocessor)
Intel 80386 iSBC 386/21 1MB memory (bare microprocessor)
Titan 8086 SECS 86/20 640KMB memory (bare microprocessor)
Titan 80286 SECS 286/20 640KMB memory (bare microprocessor)

Testing Completed 25 Nov 1987 Using ACVC 1.9

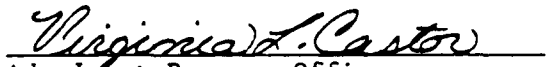
This report has been reviewed and is approved.



Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Bureau of Standards
Gaithersburg, MD 20899



Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria, VA 22311



Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the DACS-80x86, Version 4.2, using Version 1.9 of the Ada Compiler Validation Capability (ACVC). The DACS-80x86 is hosted on a DEC MicroVAX II operating under MicroVMS, Version 4.4. Programs processed by this compiler may be executed on a

Intel 8086 iSBC 86/05A 1MB memory (bare microprocessor)
Intel 80186 iSBC 186/03A 1MB memory (bare microprocessor)
Intel 80386 iSBC 386/21 1MB memory (bare microprocessor)
Titan 8086 SECS 86/20 640KMB memory (bare microprocessor)
Titan 80286 SECS 286/20 640KMB memory (bare microprocessor)

On-site testing was performed 15 Nov 1987 through 25 Nov 1987 at Phoenix, Arizona, under the direction of the Software Standards Validation Group, Institute for Computer Sciences and Technology, National Bureau of Standards (AVF), according to Ada Validation Organization (AVO) policies and procedures. At the time of testing, version 1.9 of the ACVC comprised 3122 tests of which 25 had been withdrawn. Of the remaining tests, 411 were determined to be inapplicable to this implementation. Results for processed Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 411 of the processed tests determined to be inapplicable. The remaining 2686 tests were passed. The results of validation are summarized in the following table:

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	187	489	549	247	166	98	140	326	135	36	232	3	78	2686
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	17	84	126	1	0	0	3	1	2	0	2	0	175	411
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	2	25
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

This information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of test are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

To attempt to identify any unsupported language constructs required by the Ada Standard

To determine that the implementation-dependent behavior is allowed by the Ada Standard

On-site testing was conducted from 15 Nov 1987 through 25 Nov 1987 at Phoenix, Arizona.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Software Standards Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983.

2. Ada Compiler Validation Procedures and Guidelines. Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide. SofTech, Inc., December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established procedures.
AVO	The Ada Validation Organization. In the context of this, report, the AVO is responsible for establishing procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor,

	including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Language Maintenance	The Language Maintenance Panel (LMP) is a committee established by the Ada Board to recommend interpretations and Panel possible changes to the ANSI/MIL-STD for Ada.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	An Ada program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective had been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an

Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of these units is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are

operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: DACS-80x86, Version 4.2

ACVC Version: 1.9

Certificate Number: 871125S1.09003

Host Computer:

Machine: DEC MicroVAX II

Operating System: MicroVMS
Version 4.4

Memory Size: 16 Mb RAM

Target Computer:

Machine:	Memory Size:	Operating System:
Intel 8086 iSBC 86/05A	1MB memory	(bare microprocessor)
Intel 80186 iSBC 186/03A	1MB memory	(bare microprocessor)
Intel 80386 iSBC 386/21	1MB memory	(bare microprocessor)
Titan 8086 SECS 86/20	640KMB memory	(bare microprocessor)
Titan 80286 SECS 286/20	640KMB memory	(bare microprocessor)

The disk system and other hardware components:

1 445 Mb disk drive
16 terminal ports
1 1600/6250 magnetic tape drive
1 tk50 cartridge tape drive
1 240 cps LA210 printer
1 300 dpi laser printer

Communications Network:

Intel I^2ICE (in-circuit emulator) to allow dynamic loading of test programs

Compaq 286 AT (12MHz) to host the I^2ICE
1Mb RAM
30 Mb hard disk drive
1.2 Mb read/write floppy drive

Thin-Wire Ethernet LAN between the MicroVAX and the PC to allow transfer of executable programs
DESTAA ThinWire Ethernet Station Adaptor
Transceiver cable for DEQNA --> PC connection
3COM board for Ethernet --> PC connection
DECnet-VAX running on MicroVAX II
DECnet-DOS running on PC

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler capacity is exceeded by block statements nested to 65 levels. The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See test D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

This implementation supports the additional predefined types SHORT_INTEGER, LONG_INTEGER, and LONG_FLOAT in the package STANDARD. (See tests B86001C and B86001D.)

- Based literals.

An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

Apparently all default initialization expressions or record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Apparently NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently NUMERIC_ERROR is raised when a literal operand in a fixed point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is gradual. (See tests C45524A..Z.)

- Rounding.

The method used for rounding to integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

- Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR . (See test C36003A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array objects are declared . (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant

constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with disciminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index subtype. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are not supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See test A39005C.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are not supported. (See tests A39005E and C87B62C.)

Record representation clauses are not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are not supported. (See test C87B62A.)

- Pragmas.

The pragma INLINE is supported for procedures. The pragma INLINE is supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests EE2201D, and EE2201E.)

The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests EE2401D, and EE2401G.)

There are no strings which are illegal external file names for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102C and CE2102H.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, and DIRECT_IO. (See tests CE2102G and CE2102K.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits

this behavior for SEQUENTIAL_IO, and DIRECT_IO. (See tests CE2106A and CE2106B.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO. (See test CE2208B.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for TEXT_IO. (See test EE3102C.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for TEXT_IO. (See tests CE2110B, CE2111D, CE3111A..E (5 tests), CE3114B, and CE3115A.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, and DIRECT_IO. (See tests CE2107A..D (4 tests) and CE2111D.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for DIRECT_IO. (See tests CE2107E..I (5 tests) and CE2111H.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, and DIRECT_IO. (See tests CE2108A and CE2108C.)

- Generics.

Generic subprogram declarations and bodies can compiled in separate compilations. (See test CA1012A.)

Generic package declarations and bodies can be compiled in separate compilations so long as no instantiations of those units precede the bodies. This compiler requires that a generic

unit's body be compiled prior to instantiation, and so the unit containing the instantiations is rejected.

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

At the time of testing, version 1.9 of the ACVC comprised 3122 tests of which 25 had been withdrawn. Of the remaining tests, 411 were determined to be inapplicable to this implementation. Not all of the inapplicable tests were processed during testing; 201 executable tests that use floating-point precision exceeding that supported by the implementation were not processed. Modifications to the code, processing, or grading for 77 test files were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	108	1047	1456	16	13	46	2686
Failed	0	0	0	0	0	0	0
Inapplicable	2	4	399	1	5	0	411
Withdrawn	3	2	19	0	1	0	25
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	187	489	549	247	166	98	140	326	135	36	232	3	78	2686
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	17	84	126	1	0	0	3	1	2	0	2	0	175	411
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	2	25
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 25 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C	C35904A
C35A03E	C35A03R	C37213H	C37213J	C37215C	C37215E
C37215G	C37215H	C38102C	C41402A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A	CE2401H
CE3208A					

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 411 test were inapplicable for the reasons indicated:

C24113I..K (3 tests) were rejected because they contain declarations that exceed MAX_IN_LEN (126 characters)

C34007A, **C34007D**, C34007G, C34007M, C34007P, C34007S these tests contain the application of the attribute STORAGE_SIZE to access types for which no corresponding STORAGE_SIZE length clause has been provided; this compiler rejects such an application. The AVO accepted this behavior

because the Ada standard is not clear on how such a situation should be treated; the matter will be discussed by the language maintenance body.

C35508I..J (2 tests) and C35508M..N (2 tests) use enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1). These clauses are not supported by this compiler.

C35702A uses SHORT_FLOAT which is not supported by this implementation.

A39005E and C87B62C use length clauses with SMALL specifications which are not supported by this implementation.

A39005G uses a record representation clause which is not supported by this compiler.

C45231D checks that relational and membership operations yield correct results for predefined types. This implementation supports only INTEGER, LONG_INTEGER, FLOAT, and LONG_FLOAT.

C45531M, C45531N, C45532M, and C45532N use fine 48 bit fixed point base types which are not supported by this compiler.

C45531O, C45531P, C45532O, and C45532P use coarse 48 bit fixed point base types which are not supported by this compiler.

C4A013B uses a static value that is outside the range of the most accurate floating point base type. The declaration was rejected at compile time.

D56001B uses 65 levels of block nesting which exceeds the capacity of the compiler.

B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

C87B62B this test contains the application of the attribute STORAGE_SIZE to access types for which no corresponding STORAGE_SIZE length clause has been provided; this compiler rejects such an application. The AVO accepted this behavior because the Ada standard is not clear on how such a situation should be treated; the matter will be discussed by the language maintenance body.

C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

CA2009F compiles generic subprogram declarations and bodies in separate compilations; the compilation occurs following a compilation that contains instantiations of those units. This compiler requires that a generic unit's body be compiled prior to instantiation, and so the unit containing the instantiations is rejected.

CA2009C, BC3204C, and BC3205D compile generic package specifications and bodies in separate compilations. This compiler requires that generic package specifications and bodies be in a single compilation.

EE2201D and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

EE2401D and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

The following 169 tests are inapplicable because sequential, text, and direct access files are not supported. The proper exception is raised by an attempt to create or open a sequential, text, or direct access file.

CE2104A..D(4)	CE2105A..B(2)	CE2106A..B(2)	CE2107A..I(9)
CE2108A..D(4)	CE2109A..C(3)	CE2110A..C(3)	CE2111A..E(5)
CE2111G..H(2)	CE2115A..B(2)	CE2201A..C(3)	CE2201F..G(2)
CE2204A..B(2)	CE2208B	CE2210A	CE2401A..C(3)
CE2401E..F(3)	CE2404A	CE2405B	CE2406A
CE2407A	CE2408A	CE2409A	CE2410A
CE2411A	CE3102B	EE3102C	CE3103A
CE3104A	CE3107A	CE3108A..B(2)	CE3109A
CE3110A	CE3111A..E(5)	CE3112A..B(2)	CE3114A..B(2)
CE3115A	CE3203A	CE3301A..C(3)	CE3302A
CE3305A	CE3402A..D(4)	CE3403A..C(3)	CE3403E..F(2)
CE3404A..C(3)	CE3405A..D(4)	CE3406A..D(4)	CE3407A..C(3)
CE3408A..C(3)	CE3409A	CE3409C..F(4)	CE3410A
CE3410C..F(4)	CE3411A	CE3412A	CE3413A
CE3413C	CE3602A..D(4)	CE3603A	CE3604A
CE3605A..E(5)	CE3606A..B(2)	CE3704A..B(2)	CE3704D..F(3)
CE3704M..O(3)	CE3706D	CE3706F	CE3804A..E(5)
CE3804G	CE3804I	CE3804K	CE3804M
CE3805A..B(2)	CE3806A	CE3806D..E(2)	CE3905A..C(3)
CE3905L	CE3906A..C(3)	CE3906E..F(2)	

CE2102C uses a string which is illegal as an external file name for SEQUENTIAL_IO. No illegal file names exist.

CE2102H uses a string which is illegal as an external file name for DIRECT_IO. No illegal file names exist.

The following 201 tests require a floating-point accuracy that exceeds the maximum of 15 digits supported by this implementation:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)

C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made with the approval of the AVO, and are made in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into sub-tests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 75 Class B tests, and 2 Class C tests.

The following Class B test files were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B26001A	B26002A	B26005A
B28001D	B28003A	B29001A	B2A003A
B2A003B	B2A003C	B33301A	B35101A
B37106A	B37301B	B37302A	B38003A
B38003B	B38009A	B38009B	B51001A
B53009A	B54A01C	B55A01A	B61001C
B61001D	B61001E	B61001F	B61001H
B61001I	B61001M	B61001R	B61001S
B61001W	B67001A	B67001C	B67001D
B91001A	B91002A	B91002B	B91002C
B91002D	B91002E	B91002F	B91002G
B91002H	B91002I	B91002J	B91002K
B91002L	B95030A	B95061A	B95061F
B95061G	B95077A	B97101A	B97101E
B97102A	B97103E	B97104G	BA1101B0M
BA1101B1	BA1101B2	BA1101B3	BA1101B4
BC10AEB	BC1109A	BC1109C	BC1109D
BC1202A	BC1202B	BC1202E	BC1202F
BC1202G	BC2001D	BC2001E	

The following executable tests were split because the resulting programs were too large to be executed:

C35A06N	CC1221A

The following paragraphs describe changes to the "normal" testing routines used during the ACVC on-site validation.

C4A012B checks that CONTSTRAINT_ERROR is raised for 0.0 ** (-1) or any other negative exponent value. This implementation raises NUMERIC_ERROR instead of CONSTRAINT_ERROR as permitted by LRM 4.5.5 (12) and 11.6 (7). This test was modified by the addition exception handlers for NUMERIC_ERROR and the modified test was passed. The test was run without modification and the test reported that the wrong exception was raised. The AVO ruled that either behavior (wrong exception or PASSED) is acceptable.

When D64005GOM was executed under the conditions of the RTS and basic Ada library units being pre-loaded into the low memory of each, D64005GOM raised STORAGE_ERROR. Upon further examination, it was recognized that D64005GOM produced this test result because of the run-time configuration. D64005GOM was relinked with the Run_Time System and Ada root library; when executed, D64005GOM reported PASSED. The difference between the handling of D64005GOM and the other tests was that the pre-load scheme described above used the concept that all tests could fit into a template in memory, i.e., memory was allocated for application program, stack area, heap area, etc. D64005GOM required more stack area than the other tests; it did not fit into the common template. The reserve stack area was expanded from 100 to 400 words and the main program segment size was increased from 24,000 words to 37,767 words.

All B tests were compiled on the DEC MicroVAX II host machine but were not downloaded to any target board. All L tests were compiled and linked on the DEC MicroVAX II host machine but were not downloaded to any target board. All applicable A, C, E, and F tests were compiled and linked on the DEC MicroVAX II host machine and were downloaded to each target board as described above.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the DACS-80x86 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the DACS-80x86 using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a

DEC MicroVAX II host operating under MicroVMS, Version 4.4, and the targets of:

Intel 8086 iSBC 86/05A 1MB memory (bare microprocessor)
Intel 80186 iSBC 186/03A 1MB memory (bare microprocessor)
Intel 80386 iSBC 386/21 1MB memory (bare microprocessor)
Titan 8086 SECS 86/20 640KMB memory (bare microprocessor)
Titan 80286 SECS 286/20 640KMB memory (bare microprocessor)

The host and target computers were linked via
Intel I^2ICE (in-circuit emulator) to allow dynamic loading of test programs

Compaq 286 AT (12MHz) to host the I^2ICE
1Mb RAM
30 Mb hard disk drive
1.2 Mb read/write floppy drive

Thin-Wire Ethernet LAN between the MicroVAX and the PC to allow transfer of executable programs
DESTAA ThinWire Ethernet Station Adaptor
Transceiver cable for DEQNA --> PC connection
3COM board for Ethernet --> PC connection
DECnet-VAX running on MicroVAX II
DECnet-DOS running on PC.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precision was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the DEC MicroVAX II, and all executable tests were linked and run on the:

Intel 8086 iSBC 86/05A 1MB memory (bare microprocessor)
Intel 80186 iSBC 186/03A 1MB memory (bare microprocessor)
Intel 80386 iSBC 386/21 1MB memory (bare microprocessor)
Titan 8086 SECS 86/20 640KMB memory (bare microprocessor)
Titan 80286 SECS 286/20 640KMB memory (bare microprocessor)

Object files were linked on the host computer, and executable images were transferred to the target computer via the communications network described above. DDC-I, Inc. pre-loaded the run-time system (RTS) and the basic Ada library units into each target board. This pre-loading allowed DDC-I, Inc. to link and load only the code generated for each ACVC test case (except for D64005GOM; see comment about this test below), without linking the actual RTS code into each loadable image.

An assembly file was automatically created from the operating system location map that mapped the entry points to these known locations. This file was assembled and linked with each ACVC test. The RTS and basic Ada library units were loaded into low memory on each target board before any tests were executed; each ACVC test was then loaded into a higher part of memory and executed (except for D64005GOM; see comment about this test below). The RTS and the basic Ada library units remained resident in each target board memory during the entire execution of all ACVC tests. Results were printed from the host computer, with results being transferred to the host computer via the communications network.

The compiler was tested using command scripts provided by DDC-I, Inc. and reviewed by the validation team. The compiler was tested using all default (option/switch) settings.

Tests were compiled, linked, and executed as appropriate using a single host computer and five target computers. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF.

3.7.3 Test Site

The validation team arrived at Phoenix, Arizona on 15 Nov 1987, and departed after testing was completed on 23 Nov 1987.

APPENDIX A

CONFORMANCE STATEMENT

DECLARATION OF CONFORMANCE
for Several Derived Compilers

Compiler Implementor: DDC-I, Inc.

Ada Validation Facility: Software Standards and Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, MD 20899

Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Compiler Name: DACS-80x86 Version 4.2
Host Architecture ISA: DEC MicroVax II
OS & Version #: MicroVMS 4.6
Certificate #: 871125S1.09003

Several Derived Compilers are listed below:

Derived Compiler ID: DACS-8086 Version 4.2
Host Architecture ISA: DEC Vax-11/7xx, Vax-8xxx, Vax Station, & MicroVax Series (Vax/VMS 4.6 or MicroVax/VMS 4.6)
Target Arch. ISA: Intel 8086 iSBC 86/35

Derived Compiler ID: DACS-80186 Version 4.2
Host Architecture ISA: DEC Vax-11/7xx, Vax-8xxx, Vax Station, & MicroVax Series (Vax/VMS 4.6 or MicroVax/VMS 4.6)
Target Arch. ISA: Intel 80186 iSBC 186/03A

Derived Compiler ID: DACS-80286 Version 4.2
Host Architecture ISA: DEC Vax-11/7xx, Vax-8xxx, Vax Station, & MicroVax Series (Vax/VMS 4.6 or MicroVax/VMS 4.6)
Target Arch. ISA: Intel 80286 iSBC 286/12

Derived Compiler ID:	DACS-80286 Protected Mode Version 4.2
Host Architecture ISA:	DEC Vax-11/7xx, Vax-8xxx, Vax Station, & MicroVax Series (Vax/VMS 4.6 or MicroVax/VMS 4.6)
Target Arch. ISA:	Intel 80286 iSBC 286/12 (Protected Mode)

Derived Compiler ID:	DACS-80386 Version 4.2
Host Architecture ISA:	DEC Vax-11/7xx, Vax-8xxx, Vax Station, & MicroVax Series (Vax/VMS 4.6 or MicroVax/VMS 4.6)
Target Arch. ISA:	Intel 80386 iSBC 386/21

Derived Compiler ID:	DACS-80386 Protected Mode Version 4.2
Host Architecture ISA:	DEC Vax-11/7xx, Vax-8xxx, Vax Station, & MicroVax Series (Vax/VMS 4.6 or MicroVax/VMS 4.6)
Target Arch. ISA:	Intel 80386 iSBC 386/21 (Protected Mode)

Implementor's Declaration

I, the undersigned, representing DDC-I, Inc., have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that DDC-I, Inc. is the licensor of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the licensor's name.

Lee Silverthorn

Date: May 13 1988

Lee Silverthorn, President
DDC-I, Inc.

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the DACS-80x86, Version 4.2, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A).. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is


 type INTEGER is range -32_768 .. 32_767;
 type SHORT_INTEGER is range -128 .. 127;
 type LONG_INTEGER is range  -2_147_483_648 .. 2_147_483_647;

 type FLOAT is digits 6 range
                    -3.40282366920938E38 .. 3.40282366920938E38;
 type LONG_FLOAT is digits  15 range
              -1.7976931348623157E308 .. 1.7976931348623157E308;
 type DURATION is delta 2#1.0#E-14 range -131_072.0 .. 131_071.0;



end STANDARD;
```

APPENDIX F OF THE Ada STANDARD

This appendix describes the implementation-dependent characteristics of DACS-80X86® as required in Appendix F of the Ada Reference Manual (ANSI/MIL-STD-1815A).

F.1 Implementation-Dependent Pragmas

This section describes all implementation defined pragmas.

F.1.1 Pragma INTERFACE_SPELLING

This pragma allows an Ada program to call a non-Ada program whose name contains characters that would be an invalid Ada subprogram identifier. This pragma must be used in conjunction with pragma INTERFACE, i.e., pragma INTERFACE must be specified for the non-Ada subprogram name prior to using pragma INTERFACE_SPELLING.

The pragma has the format:

```
pragma  INTERFACE_SPELLING  (subprogram name,
                             string literal);
```

where the subprogram name is that of one previously given in pragma INTERFACE and the string literal is the exact spelling of the interfaced subprogram in its native language. This pragma is only required when the subprogram name contains invalid characters for Ada identifiers.

Example:

```
function  RTS_GetDataSegment  return Integer;

pragma  INTERFACE    (ASM86, RTS_GetDataSegment);
pragma  INTERFACE_SPELLING  (RTS_GetDataSegment,
                             "R1$MGS?GetDataSegment");
```

F.1.2 Pragma INTERRUPT HANDLER

This pragma will cause the compiler to generate fast interrupt handler entries instead of the normal task calls for the entries in the task in which it is specified. It has the format:

```
pragma  INTERRUPT_HANDLER;
```

The pragma must appear as the the first thing in the specification of the task object. See section F.6.2 for more details and restrictions on specifying address clauses for task entries.

F.1.3 Pragma LT STACK SPACE

This pragma sets the size of a library task stack segment. The pragma has the format:

```
pragma  LT_STACK_SPACE (T, N);
```

where T denotes either a task object or task type and N designates the size of the library task stack segment in words.

The library task's stack segment defaults to the size of the library task stack. The size of the library task stack is normally specified via the representation clause

```
for  T'STORAGE_SIZE  use N;
```

The size of the library task stack segment determines how many tasks can be created which are nested within the library task. All tasks created within a library task will have their stacks allocated from the same segment as the library task stack. Thus, pragma LT_STACK_SPACE must be specified to reserve space within the library task stack segment so that nested tasks' stacks may be allocated.

The following restrictions are places on the use of LT_STACK_SPACE:

1) It must be used only for library tasks.

2) It must be placed immediately after the task object or type name declaration.

3) The library task stack segment size (N) must be greater than or equal to the library task stack size.

F.2 Implementation-Dependent Attributes

No implementation-dependent attributes are defined.

F.3 Package SYSTEM

The specification of the package SYSTEM:

```
package System is

   type    Word           is new Integer;
   type    LongWord       is new Long_Integer;

   type    UnsignedWord   is range 0..65535;
   for     UnsignedWord'SIZE use 16;

   subtype SegmentId      is UnsignedWord;

   type    Address is record
           offset   : UnsignedWord;
           segment  : SegmentId;
   end record;

   subtype Priority       is Word range 0..31;

   type    Name           is (iAPX86, iAPX186);

   System_Name    :       constant Name  := iAPX186;
   Storage_Unit   :       constant       := 16;
   Memory_Size    :       constant       := 1_048_576;
   Min_Int        :       constant       := -2_147_483_647-1;
   Max_Int        :       constant       :=  2_147_483_647;
   Max_Digits     :       constant       := 15;
   Max_Mantissa   :       constant       := 31;
   Fine Delta     :       constant       := 2.0 / MAX_INT;
   Tick           :       constant       := 0.000_000_125;

   type    Interface_Language is (PLM86, ASM86);

   type    ExceptionId is record
           unit_number    : UnsignedWord;
           unique_number  : UnsignedWord;
   end record;

   type    TaskValue      is new Integer;
   type    AccTaskValue   is access TaskValue;
```

```
    type    Semaphore     is
        record
            counter    :  UnsignedWord;
            first      :  TaskValue;
            last       :  TaskValue;
        end record;

    InitSemaphore   : constant Semaphore'(1, 0, 0);

end SYSTEM;
```

F.4 Representation Clauses

In general, no representation clauses may be given for a derived type. The representation clauses that are accepted for non-derived types are described in the following subsections.

F.4.1 Length Clause

Some remarks on implementation dependent behavior of length clauses are necessary:

- When using the SIZE attribute for discrete types, the maximum value that can be specified is 16 bits.
- Using the STORAGE_SIZE attribute for a collection will set an upper limit on the total size of objects allocated in this collection. If further allocation is attempted, the exception STORAGE_ERROR is raised.
- When STORAGE_SIZE is specified in a length clause for a task, the process stack area will be of the specified size. The process stack area will be allocated inside the "standard" stack segment.

F.4.2 Enumeration Representation Clause

Enumeration representation clauses may specify representations in the range of INTEGER'FIRST + 1..INTEGER'LAST - 1.

F.4.3 Record Representation Clauses

When representation clauses are applied to records the following restrictions are imposed:

- the component type is a discrete type different from LONG_INTEGER
- the component type is an array with a discrete element type different from LONG_INTEGER
- the storage unit is 16 bits
- a record occupies an integral number of storage units
- a record may take up a maximum of 32K storage units
- a component must be specified with its proper size (in bits), regardless of whether the component is an array or not.
- if a non-array component has a size which equals or exceeds one storage unit (16 bits) the component must start on a storage unit boundary, i.e. the component must be specified as:

  ```
  component     at N range 0..16 * M - 1;
  ```

  where N specifies the relative storage unit number (0,1,...) from the beginning of the record, and M the required number of storage units (1,2,...)
- the elements in an array component should always be wholly contained in one storage unit
- if a component has a size which is less than one storage unit, it must be wholly contained within a single storage unit:

  ```
  component     at N range X .. Y;
  ```

  where N is as in previous paragraph, and 0 <= X <= Y <= 15

When dealing with PACKED ARRAY the following should be noted:

- the elements of the array are packed into 1,2,4 or 8 bits

If the record type contains components which are not covered by a component clause, they are allocated consecutively after the component with the value. Allocation of a record component without a component clause is always aligned on a storage unit boundary. Holes created because of component clauses are not otherwise utilized by the compiler.

F.4.3.1 Alignment Clauses

Alignment clauses for records are implemented with the following characteristics:

- If the declaration of the record type is done at the outermost level in a library package, any alignment is accepted.
- If the record declaration is done at a given static level (higher than the outermost library level, i.e., the permanent area), only word alignments are accepted.
- Any record object declared at the outermost level in a library package will be aligned according to the alignment clause specified for the type. Record objects declared elsewhere can only be aligned on a word boundary. If the record type has been associated a different alignment, an error message will be issued.
- If a record type with an associated alignment clause is used in a composite type, the alignment is required to be one word; an error message is issued if this is not the case.

F.5 Implementation-Dependent Names for Implementation -Dependent Components

None defined by the compiler.

F.6 Address Clauses

This section describes the implementation of address clauses and what types of entities may have their address specified by the user.

F.6.1 Objects

Address clauses are supported for scalar and composite objects whose size can be determined at compile time.

F.6.2 Task Entries

The implementation supports two methods to equate a task entry to a hardware interrupt through an address clause:

1) Direct transfer of control to a task accept statement when an interrupt occurs (requires use of the pragma INTERRUPT_HANDLER).

2) Mapping of an interrupt onto a normal conditional entry call, i.e., the entry can be called from other tasks without special actions, as well as being called when an interrupt occurs.

F.6.2.1 Fast Interrupt Entry

Directly transferring control to an accept statement when an interrupt occurs requires the implementation dependent pragma INTERRUPT_HANDLER to tell the compiler that the task is an interrupt handler. By using this pragma, the user is agreeing to place certain restrictions on the task in order to speed up the software response to the hardware interrupt. Consequently, use of this method to capture interrupts is much more efficient than the general method. See section F.6.3.2.

The following constraints are placed on the task:

1) It must be a task object, i.e., not a task type.

2) The pragma must appear first in the specification of the task object.

3) All entries of the task object must be single entries with no parameters.

4) The entries must not be called from any task.

5) The body of the task object must not contain anything other than simple accept statements (potentially enclosed in a loop) referencing only global variables, i.e., no local variables. In the statement list of a simple accept statement, it is allowed to call normal, single and parameterless, entries of other tasks, but no other tasking constructs are allowed. The call to another task entry, in this case, will not lead to an immediate task context switch, but will return to the caller when complete. Once the accept is completed, the task priority rules will be obeyed, and a context switch may occur.

F.6.2.2 Normal Interrupt Entry

Mapping of an interrupt onto a normal conditional entry call puts the following constraints on the involved entries and tasks:

1) The affected entries must be defined in a task object only (not a task type).

2) The entries must be single and parameterless.

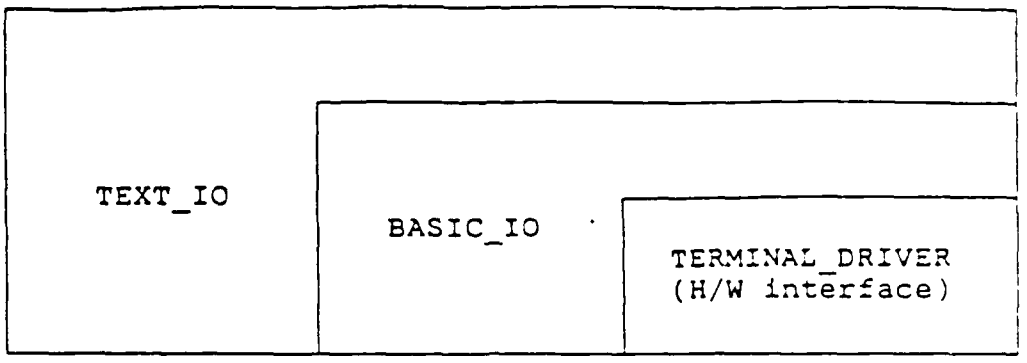
Any interrupt entry, which is not found in an interrupt handler (first method), will lead to an update of the interrupt vector segment at link time. The interrupt vector segment will be updated to point to the interrupt routine generated by the compiler to make the task entry call. The interrupt vector segment is part of the user configurable data and consists of a segment, preset to the "standard" interrupt routines (e.g., constraint_error). See section 7.2.13 (RTS Configuration of Interrupt Vector Ranges) for details on how to specify interrupt vector ranges.

F.7 Unchecked Conversions

Unchecked conversion is only allowed between objects of the same "size".

F.8 Input/Output Packages

In many embedded systems, there is no need for a traditional I/O system, but in order to support testing and validation, DDC-I has developed a small terminal oriented I/O system. This I/O system consists essentially of TEXT_IO, adapted with respect to handling only a terminal and not file I/O, and a low level package called TERMINAL_DRIVER. A BASIC_IO package has been provided for convenience purposes, forming an interface between TEXT_IO and TERMINAL_DRIVER as illustrated in the following figure:



The TERMINAL_DRIVER package is the only package that is target dependent, i.e., it is the only package that need be changed when changing communications controllers. The actual body of the TERMINAL_DRIVER is written in assembly language, but an Ada interface to this body is provided. A user can also call the terminal driver routines directly, i.e., from an assembly language routine. TEXT_IO and BASIC_IO are written completely in Ada and need not be changed.

The services provided by the terminal driver are:

1) Reading a character from the communications port.

2) Writing a character to the communications port.

The terminal driver comes in two versions: one which supports tasking, i.e., asynchronous I/O, and a version which assumes no tasking.

F.8.1 Package TEXT_IO

The specification of package TEXT_IO:

```
pragma page;
with BASIC_IO;

with IO_EXCEPTIONS;
package TEXT_IO is

   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, OUT_FILE);

   type    COUNT is range 0 .. LONG_INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
   UNBOUNDED: constant COUNT:= 0; -- line and page length

  -- max. size of an integer output field 2#....#
   subtype FIELD           is INTEGER range 0 .. 35;

   subtype NUMBER_BASE     is INTEGER range 2 .. 16;

   type TYPE_SET is (LOWER_CASE, UPPER_CASE);

pragma PAGE;
   -- File Management

   procedure CREATE (FILE  : in out FILE_TYPE;
                     MODE  : in     FILE_MODE :=OUT_FILE;
                     NAME  : in     STRING    :="";
                     FORM  : in     STRING    :=""
                    );

    procedure OPEN  (FILE  : in out FILE_TYPE;
                     MODE  : in     FILE_MODE;
                     NAME  : in     STRING;
                     FORM  : in     STRING    :=""
                    );

    procedure CLOSE (FILE  : in out FILE_TYPE;)
    procedure DELETE (FILE  : in out FILE_TYPE);
    procedure RESET (FILE  : in out FILE_TYPE; MODE : in FILE_MODE);
    procedure RESET (FILE  : in out FILE_TYPE);

    function MODE   (FILE  : in FILE_TYPE) return FILE_MODE;
    function NAME   (FILE  : in FILE_TYPE) return STRING;
    function FORM   (FILE  : in FILE_TYPE) return STRING;
```

```
    function IS_OPEN(FILE  : in FILE_TYPE return BOOLEAN;

pragma PAGE;
    -- control of default input and output files

    procedure SET_INPUT  (FILE : in FILE_TYPE);
    procedure SET_OUTPUT (FILE : in FILE_TYPE);

    function STANDARD_INPUT   return FILE_TYPE;
    function STANDARD_OUTPUT  return FILE_TYPE;

    function CURRENT_INPUT    return FILE_TYPE;
    function CURRENT_OUTPUT   return FILE_TYPE;

pragma PAGE;
   -- specification of line and page lengths

   procedure SET_LINE_LENGTH  (FILE : in FILE_TYPE; TO : in COUNT);
   procedure SET_LINE_LENGTH  (TO : in COUNT);

   procedure SET_PAGE_LENGTH  (FILE : in FILE_TYPE; TO : in COUNT);
   procedure SET_PAGE_LENGTH  (TO : in COUNT);

   function  LINE_LENGTH      (FILE : in FILE_TYPE) return COUNT;
   function  LINE_LENGTH                            return COUNT;

   function  PAGE_LENGTH      (FILE : in FILE_TYPE) return COUNT;
   function  PAGE_LENGTH                            return COUNT;


pragma PAGE;
   -- Column, Line, and Page Control

   procedure NEW_LINE (FILE : in FILE_TYPE;
                       SPACING : in POSITIVE_COUNT := 1);
   procedure NEW_LINE (SPACING : in POSITIVE_COUNT := 1);

   procedure SKIP_LINE (FILE : in FILE_TYPE;
                        SPACING : in POSITIVE_COUNT := 1);
   procedure SKIP_LINE (SPACING : in POSITIVE_COUNT := 1);

   function  END_OF_LINE (FILE : in FILE_TYPE) return BOOLEAN;
   function  END_OF_LINE                       return BOOLEAN;

   procedure NEW_PAGE    (FILE : in FILE_TYPE);
   procedure NEW_PAGE                         ;

   procedure SKIP_PAGE   (FILE : in FILE_TYPE);
   procedure SKIP_PAGE                        ;
```

```
    function  END_OF_PAGE (FILE : in FILE_TYPE) return BOOLEAN;
    function  END_OF_PAGE                       return BOOLEAN;

    function  END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
    function  END_OF_FILE                       return BOOLEAN;

    procedure SET_COL     (FILE : in FILE_TYPE;
                           TO   : in POSITIVE_COUNT);
    procedure SET_COL     (TO : in POSITIVE_COUNT);

    procedure SET_LINE    (FILE : in FILE_TYPE;
                           TO : in POSITIVE_COUNT);
    procedure SET_LINE    (TO : in POSITIVE_COUNT);

    function  COL         (FILE : in FILE_TYPE) return POSITIVE_COUNT;
    function  COL                               return POSITIVE_COUNT;

    function  LINE        (FILE : in FILE_TYPE) return POSITIVE_COUNT;
    function  LINE                              return POSITIVE_COUNT;

    function  PAGE        (FILE : in FILE_TYPE) return POSITIVE_COUNT;
    function  PAGE                              return POSITIVE_COUNT;

pragma PAGE;
   -- Character Input-Output

    procedure GET  (FILE : in FILE_TYPE; ITEM : out CHARACTER);
    procedure GET  (                     ITEM : out CHARACTER);
    procedure PUT  (FILE : in FILE_TYPE; ITEM :  in CHARACTER);
    procedure PUT  (                     ITEM :  in CHARACTER);

    -- String Input-Output

    procedure GET  (FILE : in FILE_TYPE; ITEM : out CHARACTER);
    procedure GET  (                     ITEM : out CHARACTER);
    procedure PUT  (FILE : in FILE_TYPE; ITEM :  in CHARACTER);
    procedure PUT  (                     ITEM :  in CHARACTER);

    procedure GET_LINE   (FILE : in FILE_TYPE;
                          ITEM : out STRING;
                          LAST : out NATURAL);

    procedure GET_LINE   (ITEM : out STRING;
                          LAST : out NATURAL);

    procedure PUT_LINE   (FILE : in FILE_TYPE; ITEM :  in STRING);
    procedure PUT_LINE   (                     ITEM :  in STRING);


pragma PAGE;
```

```
-- Generic Package for Input-Output of Integer Types

generic
    type NUM is range <>;
package INTEGER_IO is

   DEFAULT_WIDTH : FIELD       := NUM'WIDTH;
   DEFAULT_BASE  : NUMBER_BASE :=        10;

   procedure GET  (FILE  : in FILE_TYPE;
                   ITEM  : out NUM;
                   WIDTH : in FIELD := 0);
   procedure GET  (ITEM  : out NUM;
                   WIDTH : in FIELD := 0);

   procedure PUT  (FILE  : in FILE_TYPE;
                   ITEM  :  in NUM;
                   WIDTH :  in FIELD := DEFAULT_WIDTH;
                   BASE  :  in NUMBER_BASE := DEFAULT_BASE);
   procedure PUT  (ITEM  :  in NUM;
                   WIDTH :  in FIELD := DEFAULT_WIDTH;
                   BASE  :  in NUMBER_BASE := DEFAULT_BASE);

   procedure GET  (FROM  : in STRING;
                   ITEM  : out NUM;
                   LAST  : out POSITIVE);

   procedure PUT  (TO    : out STRING;
                   ITEM  :  in NUM;
                   BASE  :  in NUMBER_BASE := DEFAULT_BASE);

end INTEGER_IO;

pragma PAGE;
```

```
   -- Generic Packages for Input-Output of Real Types

generic
   type NUM is digits <>;
package FLOAT_IO is

   DEFAULT_FORE : FIELD :=               2;
   DEFAULT_AFT  : FIELD := NUM'DIGITS - 1;
   DEFAULT_EXP  : FIELD :=               3;

   procedure GET  (FILE  : in FILE_TYPE;
                   ITEM  : out NUM;
                   WIDTH : in FIELD := 0);
   procedure GET  (ITEM  : out NUM;
                   WIDTH : in FIELD := 0);

   procedure PUT  (FILE : in FILE_TYPE;
                   ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);
   procedure PUT  (ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure GET  (FROM : in STRING;
                   ITEM : out NUM;
                   LAST : out POSITIVE);
   procedure PUT  (TO   : out STRING;
                   ITEM :  in NUM;
                   AFT  :  in FIELD := DEFAULT_AFT;
                   EXP  :  in FIELD := DEFAULT_EXP);

end FLOAT_IO;


pragma PAGE;
```

```
generic
   type NUM is delta <>;
package FIXED_IO is

   DEFAULT_FORE : FIELD := NUM'FORE;
   DEFAULT_AFT  : FIELD := NUM'AFT;
   DEFAULT_EXP  : FIELD := 0;

   procedure GET  (FILE  : in FILE_TYPE;
                   ITEM  : out NUM;
                   WIDTH : in FIELD := 0);
   procedure GET  (ITEM  : out NUM;
                   WIDTH : in FIELD := 0);

   procedure PUT  (FILE : in FILE_TYPE;
                   ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);
   procedure PUT  (ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure GET  (FROM : in STRING;
                   ITEM : out NUM;
                   LAST : out POSITIVE);

   procedure PUT  (TO   : out STRING;
                   ITEM :  in NUM;
                   AFT  :  in FIELD := DEFAULT_AFT;
                   EXP  :  in FIELD := DEFAULT_EXP);

end FIXED_IO;


pragma PAGE;
```

```
   -- Generic Package for Input-Output of Enumeration Types

generic
   type ENUM is (<>);
package ENUMERATION_IO is

   DEFAULT_WIDTH   : FIELD    := 0;
   DEFAULT_SETTING : TYPE_SET := UPPER_CASE;

   procedure GET  (FILE : in FILE_TYPE; ITEM : out ENUM);
   procedure GET  (                     ITEM : out ENUM);

   procedure PUT  (FILE  : FILE_TYPE;
                   ITEM  : in ENUM;
                   WIDTH : in FIELD      := DEFAULT_WIDTH;
                   SET   : in TYPE_SET   := DEFAULT_SETTING);
   procedure PUT  (ITEM  : in ENUM;
                   WIDTH : in FIELD      := DEFAULT_WIDTH;
                   SET   : in TYPE_SET   := DEFAULT_SETTING);

   procedure GET  (FROM : in  STRING;
                   ITEM : out ENUM;
                   LAST : out POSITIVE);

   procedure PUT  (TO   : out STRING;
                   ITEM : in  ENUM;
                   SET  : in  TYPE_SET := DEFAULT_SETTING);

end ENUMERATION_IO;


pragma PAGE;

   -- Exceptions

   STATUS_ERRO  : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

pragma page;
private

   type FILE_TYPE is new interger;

end TEXT_IO;
```

F.8.2 Package IO_EXCEPTIONS

The specification of the package IO_EXCEPTIONS:

```
package IO_EXCEPTIONS is

   STATUS_ERROR : exception;
   MODE_ERROR   : exception;
   NAME_ERROR   : exception;
   USE_ERROR    : exception;
   DEVICE_ERROR : exception;
   END_ERROR    : exception;
   DATA_ERROR   : exception;
   LAYOUT_ERROR : exception;

end IO_EXCEPTIONS;
```

F.8.3 Package BASIC_IO

The specification of package BASIC_IO:

```
with IO_EXCEPTIONS;

package BASIC_IO is

   type count is range 0 .. integer'last;

   subtype positive_count is count range 1 .. count'last;


   function get_integer return string;

   -- Skips any leading blanks, line terminators or page
   -- terminators. Then reads a plus or a minus sign if
   -- present, then reads according to the syntax of an
   -- integer literal, which may be based.  Stores in item
   -- a string containing an optional sign and an integer
   -- literal.
   --
   -- The exception DATA_ERROR is raised if the sequence
   -- of characters does not correspond to the syntax
   -- described above.
   --
   -- The exception END_ERROR is raised if the file terminator
   -- is read.  This means that the starting sequence of an
   -- integer has not been met.
   --
   -- Note that the character terminating the operation must
   -- be available for the next get operation.
   --
```

```
function get_real return string;

-- Corresponds to get_integer except that it reads according
-- to the syntax of a real literal, which may be based.


function get_enumeration return string;

-- Corresponds to get_integer except that it reads according
-- to the syntax of an identifier, where upper and lower
-- case letters are equivalent to a character literal
-- including the apostrophes.


function get_item (length : in     integer) return string;

-- Reads a string from the current line and stores it in
-- item.  If the remaining number of characters on the
-- current line is less than length then only these
-- characters are returned.  The line terminator is not
-- skipped.


procedure put_item (item : in     string);

-- If the length of the string is greater than the current
-- maximum line (linelength),  the exception LAYOUT_ERROR
-- is raised.
--
-- If the string does not fit on the current line a line
-- terminator is output, then the item is output.


-- Line and page lengths - ARM 14.3.3.
--

procedure set_line_length (to   : in     count);


procedure set_page_length (to   : in     count);


function line_length return count;


function page_length return count;
```

```
-- Operations on columns, lines and pages - ARM 14.3.4.
--

procedure new_line;

procedure skip_line;

function end_of_line return boolean;

procedure new_page;

procedure skip_page;

function end_of_page return boolean;

function end_of_file return boolean;

procedure set_col (to   : in    positive_count);

procedure set_line (to   : in     positive_coun );

function col return positive_count;

function line return positive_count;

function page return positive_count;
```

```
    -- Character and string procedures.
    -- Corresponds to the procedures defined in ARM 14.3.6.
    --

    procedure get_character (item :    out character);


    procedure get_string (item :    out string);


    procedure get_line (item :    out string;
                        last :    out natural);


    procedure put_character (item : in     character);


    procedure put_string (item : in     string);


    procedure put_line (item : in     string);


  --  exceptions:

  USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
  DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
  END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
  DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
  LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

end BASIC_IO;
```

F.8.4 Package LOW LEVEL IO

The specification of LOW_LEVEL_IO is:

```
with SYSTEM;

package LOW_LEVEL_IO is

        subtype port_address is System.Word;

        type byte is new integer;

        procedure send_control(device : in port_address;
                               data   : in System.Word);

        procedure send_control(device : in port_address;
                               data   : in byte);

        procedure recieve_control(device : in     port_address;
                                  data   :    out byte);

        procedure receive_control(device : in     port_address;
                                  data   :    out System.Word);

        private

                pragma(inline(send_control, receive_control);

end LOW_LEVEL_IO;
```

F.8.5 Package TERMINAL_DRIVER

The specification of package TERMINAL_DRIVER:

```
package TERMINAL_DRIVER is

  procedure put_character (ch :  in  character);

  procedure get_character (ch : out  character);

private

  pragma interface (ASM86, put_character);

  pragma interface (ASM86, get_character);

end TERMINAL_DRIVER;
```

F.9 Machine Code Insertions

The reader should be familiar with the code generation strategy and the 8086/80186 instruction set to fully benefit from this chapter.

As described in chapter 13.8 of the ARM [DoD 83] it is possible to write procedures containing only code statements using the predefined package MACHINE_CODE. The package MACHINE_CODE defines the type MACHINE_INSTRUCTION which, used as a record aggregate, defines a machine code insertion. The following sections list the type MACHINE_INSTRUCTION and types on which it depends, give the restrictions, and show an example of how to use the package MACHINE_CODE.

F.9.1 Predefined Types for Machine Code Insertions

The following types are defined for use when making machine code insertions (their type declarations are given in the following pages):

```
type  opcode_type
type  operand_type
type  register_type
type  segment_register
type  machine_instruction
```

The type REGISTER_TYPE defines registers and register combinations. The double register combinations (e.g. BX_SI) can be used only as address operands (BX_SI describing [BX+SI]). The registers STi describe registers on the floating stack. (ST is the top of the floating stack).

The type SEGMENT_REGISTER defines the four segment registers that can be used to overwrite default segments in an address operand.

The type MACHINE_INSTRUCTION is a discriminant record type with which every kind of instruction can be described. Symbolic names may be used in the form

```
name'ADDRESS
```

Restrictions as to symbolic names can be found in section F.9.2.

```
type opcode_type is (   m_AAA,      m_AAD,      m_AAM,      m_AAS,
                        m_ADC,      m_ADD,      m_AND,      m_ARPL,
                        m_BOUND,    m_CALL,     m_CBW,      m_CLC,
                        m_CLD,      m_CLI,      m_CLTS,     m_CMC,
                        m_CMP,      m_CMPS,     m_CMPSB,    m_CMPSW,
                        m_CWD,      m_DAA,      m_DAS,      m_DEC,
                        m_DIV,      m_ENTER,    m_HLT,      m_IDIV,
                        m_IMUL,     m_IN,       m_INC,      m_INS,
                        m_INSB,     m_INSW,     m_INT,      m_INTO,
                        m_IRET,     m_JA,       m_JAE,      m_JB,

                        m_JBE,      m_JC,       m_JCXZ,     m_JE,
                        m_JG,       m_JGE,      m_JL,       m_JLE,
                        m_JNA,      m_JNAE,     m_JNB,      m_JNBE,
                        m_JNC,      m_JNE,      m_JNG,      m_JNGE,
                        m_JNL,      m_JNLE,     m_JNO,      m_JNP,
                        m_JNS,      m_JNZ,      m_JO,       m_JP,
                        m_JPE,      m_JPO,      m_JS,       m_JZ,
                        m_JMP,      m_LAHF,     m_LAR,      m_LDS,
                        m_LES,      m_LEA,      m_LEAVE,    m_LGDT,
                        m_LIDT,     m_LLDT,     m_LMSW,     m_LOCK,

                        m_LODS,     m_LODSB,    m_LODSW,    m_LOOP,
                        m_LOOPE,    m_LOOPNE,   m_LOOPNZ,   m_LOOPZ,
                        m_LSL,      m_LTR,      m_MOV,      m_MOVS,
                        m_MOVSB,    m_MOVSW,    m_MUL,      m_NEG,
                        m_NOP,      m_NOT,      m_OR,       m_OUT,
                        m_OUTS,     m_OUTSB,    m_OUTSW,    m_POP,
                        m_POPA,     m_POPF,     m_PUSH,     m_PUSHA,
                        m_PUSHF,    m_RCL,      m_RCR,      m_ROL,
                        m_ROR,      m_REP,      m_REPE,     m_REPNE,
                        m_RET,      m_SAHF,     m_SAL,      m_SAR,
                        m_SHL,      m_SHR,      m_SBB,      m_SCAS,

                        m_SCASB,    m_SCASW,    m_SGDT,     m_SIDT,
                        m_SLDT,     m_SMSW,     m_STC,      m_STD,
                        m_STI,      m_STOS,     m_STOSB,    m_STOSW,
                        m_STR,      m_SUB,      m_TEST,     m_VERR,
                        m_VERW,     m_WAIT,     m_XCHG,     m_XLAT,
                        m_XOR,
```

```
m_FABS,     m_FADD,     m_FADDD,    m_FADDP,
m_FBLD,     m_FBSTP,    m_FCHS,     m_FNCLEX,
m_FCOM,     m_FCOMD,    m_FCOMP,    m_FCOMPD,
m_FCOMPP,   m_FDECSTP,  m_FDIV,     m_FDIVD,
m_FDIVP,    m_FDIVR,    m_FDIVRD,   m_FDIVRP,
m_FFREE,    m_FIADD,    m_FIADDD,   m_FICOM,
m_FICOMD,   m_FICOMP,   m_FICOMPD,  m_FIDIV,
m_FIDIVD,   m_FIDIVR,   m_FIDIVRD,  m_FILD,
m_FILDD,    m_FILDL,    m_FIMUL,    m_FIMULD,
m_FINCSTP,  m_FNINIT,   m_FIST,     m_FISTD,

m_FISTP,    m_FISTPD,   m_FISTPL,   m_FISUB,
m_FISUBD,   m_FISUBR,   m_FISUBRD,  m_FLD,
m_FLDD,     m_FLDCW,    m_FLDENV,   m_FLDLG2,
m_FLDLN2,   m_FLDL2E,   m_FLDL2T,   m_FLDPI,
m_FLDZ,     m_FLD1,     m_FMUL,     m_FMULD,
m_FMULP,    m_FNOP,     m_FPATAN,   m_FPREM,
m_FPTAN,    m_FRNDINT,  m_FRSTOR,   m_FSAVE,
m_FSCALE,   m_FSETPM,   m_FSQRT,    m_FST,
m_FSTD,     m_FSTCW,    m_FSTENV,   m_FSTP,
m_FSTPD,    m_FSTSW,    m_FSTSWAX,  m_FSUB,
m_FSUBD,    m_FSUBP,    m_FSUBR,    m_FSUBRD,

m_FSUBRP,   m_FTST,     m_FWAIT,    m_FXAM,
m_FXCH,     m_FXTRACT,  m_FYL2X,    m_FYL2XP1,
m_F2XM1,    m_label,    m_reset);
```

```
type operand_type is (none,                    -- no operands

                      immediate,               -- 1 immediate operand

                      register,                -- 1 register operand

                      address,                 -- 1 address operand

                      system_address,          -- 1 'address operand

                      register_immediate,      -- 2 operands:  dest is
                                               -- register, source is
                                               -- immediate

                      register_register,       -- 2 register operands

                      register_address,        -- 2 operands: dest is
                                               -- register, source is
                                               -- address

                      address_register,        -- 2 operands: dest is
                                               -- address, source is
                                               -- register

                      register_system_address, -- 2 operands: dest is
                                               -- register, source is
                                               -- 'address

                      system_address_register, -- 2 operands: dest is
                                               -- 'address, source is
                                               -- register

                      address_immediate,       -- 2 operands: dest is
                                               -- 'address, source is
                                               -- immediate

                      system_address_immediate,-- 2 operands: dest is
                                               -- 'address, source is
                                               -- immediate

                      immediate_register,      -- only allowed for
                                               -- OUT
                                               -- port is immediate
                                               -- source is register

                      immediate_immediate);    -- only allowed for
                                               -- ENTER
```

```
type register_type is (AX, CX, DX, BX, SP, BP, SI, DI, -- registers
                                                       -- and
                                                       -- possible

                       AL, CL, DL, BL, AH, CH, DH, BH, -- register
                                                       -- combina-
                                                       -- tions
                       ES, CS, SS, DS,

                       BX_SI,  BX_DI,  BP_SI,  BP_DI,

                       ST,     ST1,    ST2,    ST3,    -- floating
                                                       -- stack
                                                       -- registers

                       ST4,    ST5,    ST6,    ST7,

                       nil );


type segment_register is ( ES, CS, SS, DS, nil );      -- segment
                                                       -- registers
```

```
type machine_instruction (operand_kind : operand_type is
   record
      opcode : opcode_type;

      case operand_kind is
         when immediate =>
            immediate          : integer;

         when register =>
            register1          : register_type;

         when address =>
            segment1           : segment_register;
            address_register1  : register_type;
            offset1            : integer;

         when system_address =>
            addr1              : system.address;

         when register_immediate =>
            register2          : register_type;
            immediate2         : integer;

         when register_register =>
            register3          : register_type;
            register4          : register_type;

         when register_address =>
            register5          : register_type;
            segment2           : segment_register;
            address_register2  : register_type;
            offset2            : integer;

         when address_register =>
            segment3           : segment_register;
            address_register3  : register_type;
            offset3            : integer;
            register6          : register_type;

         when register_system_address =>
            register7          : register_type;
            addr2              : system.address

         when system_address_register =>
            addr3              : system.address;
            register8          : register_type;
```

```
      when address_immediate =>
         segment4          : segment_register;
         address_register4 : register_type;
         offset4           : integer;
         immediate3        : integer;

      when system_address_immediate =>
         addr4             : system.address;
         immediate4        : integer;

      when immediate_register =>
         immediate5        : integer;
         register9         : register_type;

      when immediate_immediate =>
         immediate6        : integer;
         immediate7        : integer;

      when others =>
         null;
   end case;
end record;
```

F.9.2 Restrictions

Only procedures, and not functions, may contain machine code insertions.

Symbolic names in the form x'ADDRESS can only be used in the following cases:

1) x is an object of scalar type or access type declared as an object, a formal parameter, or by static renaming.

2) x is an array with static constraints declared as an object (not as a formal parameter or by renaming).

3) x is a record declared as an object (not a formal parameter or by renaming).

All opcodes defined by the type OPCODE_type except the m_CALL can be used.

Two opcodes to handle labels have been defined:

m_label: defines a label. The label number must be in the range 1 <= x <= 25 and is put in the offset field in the first operand of the MACHINE_INSTRUCTION.

m_reset: used to enable use of more than 25 labels. The label number after a m_RESET must be in the range 1 <= x <=25. To avoid errors you must make sure that all used labels have been defined before a reset, since the reset operation clears all used labels.

All floating instructions have at most one operand which can be any of the following:

- a memory address
- a register or an immediate value
- an entry in the floating stack

When entering a procedure with machine code insertions, BP has been placed at the top of the stack, and parameters for this procedure are placed in [BP-2] .. [BP-n].

F.9.3 Examples

The following section contains examples of how to use the machine code insertions and lists the generated code.

F.9.3.1 Example Using Labels

The following assembler code can be described by machine code insertions as shown:

```
     MOV AX,7
     MOV CX,4
     CMP AX,CX
     JG  1
     JE  2
     MOV CX,AX
  1: ADD AX,CX
  2: MOV SS: [BP+DI], AX
```

```
with MACHINE_CODE; use MACHINE_CODE;
procedure test_labels is

begin

  MACHINE_INSTRUCTION'(register_immediate, m_MOV, AX, 7);

  MACHINE_INSTRUCTION'(register_immediate, m_MOV, CX, 4);

  MACHINE_INSTRUCTION'(register_register, m_CMP, AX, CX);

  MACHINE_INSTRUCTION'(immediate, m_JG, 1);

  MACHINE_INSTRUCTION'(immediate, m_JE, 2);

  MACHINE_INSTRUCTION'(register_register, m_MOV, CX, AX);

  MACHINE_INSTRUCTION'(immediate, m_LABEL, 1);

  MACHINE_INSTRUCTION'(register_register, m_ADD, AX, CX);

  MACHINE_INSTRUCTION'(immediate, m_label, 2);

  MACHINE_INSTRUCTION'(address_register, m_MOV, SS, BP_DI, 0, AX);

end test_labels;
```

F.9.3.2 Example Using Symbolic Names

The following procedure will add two integers and return the result in the last parameter.

```
  with MACHINE_CODE;              Use MACHINE_CODE;

  procedure mk_add
           (a : in     integer:,
            b : in     integer:,
            c :    out integer)

  Begin

machine_instruction'(register_system_address,m_MOV,AX,a'address);


machine_instruction'(register_system_address,m_ADD,AX,b'address);

    machine_instruction'(none, m_INTO);

    machine_instruction'(system_address_register, m_MOV,
                         c'address, AX);

  end mk_add
```

The generated assembler code will be:

```
   MOV  AX, [BP-6]
   ADD  AX, [BP-4]
   INTO
   MOV  [BP-2], AX
```

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	<125 X "A">1
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	<125 X "A">2
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	<63 X "A">3<62 X "A">
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	<63 X "A">4<62 X "A">
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	<123 X "0">298
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	<120 X "0">69.0E1

$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	<63 X "A">
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	<62 X "A">1
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	<106 X " ">
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	32_767
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	35
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	BAD-CHARS^#.%!X
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	WILD-CHAR*.NAM
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	200_000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	ILLEGAL!@#$%^

$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	ILLEGAL&()_+=
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32_768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32_767
$1NTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32_768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-200_000.0
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	126
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2_147_483_647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	2:<121 X "0">11:

$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	16:<119 X "0">F.E:
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	"<124 X "A">"
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_ INT.	-2_147_483_648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFF#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 25 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

B28003A A basic declaration (line 36) wrongly follows a later declaration.

E28005C This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ALMP.

C34004A The expression in line 168 wrongly yield a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

C35502P The equality operators in lines 62 and 69 should be inequality operators

A35902C Line 17's assignment of the nominal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

C35904A The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

C35A03E & R These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

C37213H The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

C37213J The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

C37215C, E,G,H Various discriminant constraints are wrongly expected to be incompatible with the type CONS.

C38102C The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

C41402A 'STORAGE_SIZE is wrongly applied to an object of an access type.

C45614C REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).

A74106C A bound specified in a fixed-point subtype declaration lies
C85018B outside of that calculated for the base type, raising
C87B04B CONSTRAINT_ERROR. Errors of this sort occur about lines 37 &
CC1311B 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively (and possibly elsewhere)

BC3105A Lines 159..168 are wrongly expected to be incorrect; they are correct.

AD1A01A The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.

CE2401H The record aggregates in lines 105 and 117 contain the wrong values.

CE3208A This test expects that an attempt to open the default output file (after it was closed) with MODE_IN file raises NAME_ERROR or USE_ERROR; by commentary AI-00048, MODE_ERROR should be raised.